어드벤처디자인 기초 레벨

세그먼트 트리를 사용한

영역 합의 질의 문제

|  |  |
| --- | --- |
| 학과 | 컴퓨터공학과 |
| 학번 | 20221049 |
| 학년 | 2 |
| 이름 | 정보경 |
| 제출 일자 | 2023. 09. 22. |

목차

1. 문제 개요
2. 문제 해결 방법
3. 알고리즘 분석
4. 구현 및 실험 결과
5. 기타 사항

[붙임] 소스코드

1. 문제 개요

n개의 정수 값을 갖는 배열 list[0], list[1], list[2], … , list[n-1]이 주어질 때 주어진 list 배열에서 인덱스 s에서 e까지의 배열 원소들의 합을 구한다. 여기서 인덱스 s에서 e의 범위는 0≤s≤e≤n-1이 된다. 이러한 문제를 영역 질의 합의(range sum query) 문제라 한다.

그러나 이 문제를 단순하게 s부터 e까지 반복문을 통해 더하게 되면 수행시간은 O(n)이 된다. 따라서 이 문제의 수행시간을 개선하기 위한 자료구조를 채택하여야 한다

2. 문제 해결 방법

수행시간을 개선하기 위한 자료구조로 세그먼트 트리를 사용한다. 세그먼트 트리는 배열로 주어진 입력 리스트의 원소들의 부분 합을 이진 트리 형식으로 구성한 자료구조이다. 입력된 리스트를 반으로 쪼개어 두 개의 리스트로 분할하고, 또 그 각각의 분할된 리스트를 다시 두 개의 리스트로 분할한다. 마지막에 각 원소로 모두 쪼개져 더 이상 분할할 수 없는 상태가 되면, 각 원소들은 단말 노드가 되어 트리를 구성한다. 이렇게 구성된 단말 노드부터 시작하여 부모 노드는 자식 원소의 합이 되며 트리의 비 단말 노드를 구성해 나간다. 즉, 부모 노드는 자식 노드의 합이고, 그 부모 노드 또한 해당 노드의 형제 노드와의 합이 부모 노드로 저장되는 형태를 이루는 것이다. 또한 세그먼트 트리는 배열을 사용한다. 일반적으로, 완전이진트리에서 배열을 사용하고, 일반적인 이진트리에서는 연결리스트를 사용한다. 세그먼트 트리는 완전이진트리는 아니지만, 매번 반으로 쪼개어 트리를 구성하므로 균형 잡힌 트리라 볼 수 있다. 따라서 배열을 사용하여 구성한다.

이러한 세그먼트 트리를 사용하기 위해 아래 세 가지의 알고리즘을 설계한다.

* 입력 리스트의 배열을 세그먼트 트리로 구성한다.
* 세그먼트 트리를 사용하여 질의 범위의 합을 계산한다.
* 입력 리스트의 배열이 변경될 경우 세그먼트 트리를 갱신한다.

3. 알고리즘 분석

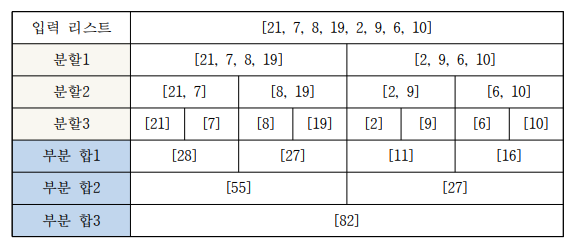
세그먼트 트리의 구성에서 n개의 입력 리스트가 단말 노드에 배치되고 부분 합을 저장하기 위해 이를 이진 트리로 표현하면 트리의 높이는 logn이 된다. 따라서 세그먼트 트리에서의 연산은 수행시간이 O(logn)이 된다. 반면에 포화이진트리의 노드 수는 2 × 2트리의 높이 - 1 이므로 세그먼트 트리를 저장하기 위해 배열 공간을 2 × 2⌈logn⌉ - 1의 크기로 할당하게 된다. 따라서 수행공간은 O(2⌈logn⌉)이 된다.

**(1) 세그먼트 트리의 구성**

주어진 입력 리스트를 같은 크기 혹은 한 개의 원소만큼만 차이 나는 크기의 두 리스트로, 나눈 리스트를 마찬가지로 두 개의 리스트로 분할한다. 이러한 분할은 최종적으로 더 이상 분할 되지 않는 상태, 즉 한 개의 원소만 남아있을 때까지 반복한다.

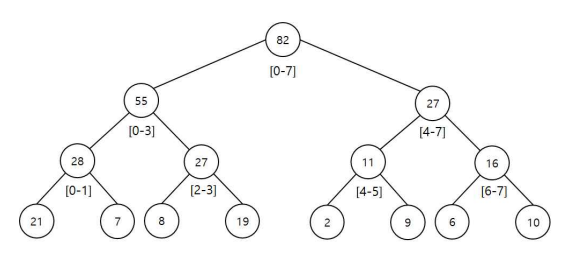
분할이 끝나면 부분합을 구성한다. 분할되어 한 개의 원소로 이루어진 리스트들을 단말 노드로 구성한 상태로, 비 단말 노드는 자신의 자식의 합이 되도록 구성한다. 형제 노드와의 합을 부모 노드에 기록하고, 그 부모 노드 또한 형제 노드와의 합을 자신의 부모 노드에 기록하며 루트 노드까지 반복하게 된다. 즉 단말 노드에서부터 루트 노드까지 반복하여 합을 채워 넣는 방식으로 구성하게 된다. (표 1.1)은 이러한 세그먼트 트리의 구성 과정을 보여준다.

(표 1.1) 입력 리스트를 분할한 후에 세그먼트 트리를 구성하는 과정



(그림 1.2)는 (표 1.1)의 구성 과정을 통해 만들어진 세그먼트 트리이다. 단말 노드는 입력 리스트의 각 원소들이 배치되었으며, 비 단말 노드는 각각의 두 자식의 합이 저장되었다. 세그먼트 트리를 구성하기 위해 리스트의 분할과 부분 합의 계산을 위한 두 개의 단계로 구분되며, 입력 리스트에 포함된 데이터의 수가 n일 경우 각각 O(logn)번이 반복 수행된다.

(그림 1.2) 8개의 입력 리스트에 대한 세그먼트 트리의 구성



(알고리즘 Seg\_tree\_const)는 n개의 정수형 데이터를 갖는 입력 리스트가 배열 list로 주어질 경우 (알고리즘 Construct\_tree)를 호출하여 세그먼트 트리를 구성하는 알고리즘이다. 알고리즘의 입력은 n개의 정숫값이 저장된 list 배열이고 출력은 세그먼트 트리가 구성된 배열 seg\_tree이다.

1. Algorithm Seg\_tree\_const(list, n)
2. Input: n개의 정수형 데이터를 갖는 입력 리스트 배열(list)
3. Output: 입력 리스트의 세그먼트 트리를 저장한 배열(seg\_tree)
4. height <- ⌈logn⌉
5. t\_size <- 2 × 2height - 1
6. seg\_tree <- malloc(t\_size)
7. Construct\_tree(list, 0, n-1, seg\_tree, 0)
8. return seg\_tree

(알고리즘 Construct\_tree)는 입력 리스트를 반복적으로 분할하고 마지막으로 분할된 리스트로부터 부분 합을 계산하여 세그먼트 트리를 구성하는 알고리즘이다. 알고리즘의 입력에서 start와 end는 가장 최근에 분할된 리스트의 시작과 끝 인덱스 값을 의미한다. Current는 재귀적으로 반복되는 과정에서 부분 합을 계산하기 위해 현재의 상태에서 세그먼트 트리의 배열 seg\_tree의 인덱스 값을 의미한다.

1. Algorithm Construct\_tree(list, start, end, seg\_tree, current)
2. Input: 입력 리스트 배열(list), 분할된 리스트의 시작(start)과 끝(end), 현재 노드의

인덱스(current), 세그먼트 트리 배열(seg\_tree)

1. Output: 세그먼트 트리의 현재 인덱스 배열
2. if (start = end) {
3. seg\_tree[current] <- list[start]
4. return list[start] }
5. mid <- start + (end – start) / 2
6. child <- 2 \* current
7. seg\_tree[current] <- Construct\_tree(list, start, mid, seg\_tree, child+1)

+ Construct\_tree(list, mid+1, end, seg\_tree, child+2)

1. return seg\_tree[current]

**(2) 세그먼트 트리를 사용하여 질의 영역의 합 계산**

주어진 입력 리스트에 대해 (1)에서 설명한 알고리즘에 따라 구성된 세그먼트 트리를 사용하여 질의 영역의 합을 계산하는 알고리즘을 설계한다.

(알고리즘 Get\_query)는 입력 리스트에서 질의 범위로 주어진 합을 계산하여 반환하는 알고리즘이다. 알고리즘에서 입력으로 (1)에서 부분 합이 계산된 세그먼트 트리 배열(seg\_tree)과 부분 합을 계산하기 위한 질의의 시작(q\_s)과 끝(q\_e)인덱스가 주어진다. 질의로 주어진 인덱스의 범위가 올바른지 확인하여 오류 처리를 하고 종료하는 것이 4번 줄에 나타나 있다. 5번 줄에서는 (알고리즘 Query\_sum)을 호출하여 부분합의 계산 결과를 저장하고, 이를 반환에 사용한다.

1. Algorithm Get\_query(seg\_tree, n, q\_s, q\_e)
2. Input: 세그먼트 트리 배열(seg\_tree), 부분 합을 계산하기 위한 시작과 끝 인덱스

위치(q\_s, q\_e)

1. Output: 질의한 위치의 부분 합
2. if (q\_s < 0 or q\_e > n-1 or q\_e > q\_s) return error
3. sum <- Query\_sum(seg\_tree, 0, n-1, q\_s, q\_e, current)
4. return sum

(알고리즘 Query\_sum)은 입력 리스트로 구성된 세그먼트 트리의 배열을 사용하여 질의 범위의 부분 합을 계산하는 알고리즘이다. 이 알고리즘은 질의 범위의 부분 합을 계산하기 위해 다음과 같이 세 가지 경우로 구분되어 수행된다.

* 질의한 인덱스의 시작과 끝의 범위가 현재 분할된 리스트의 시작과 끝의 인덱스 범위를 포함하는 경우 세그먼트 트리 배열에서 현재 위치의 부분 합인 seg\_tree[current] 값을 반환한다(4번 줄).
* 질의한 인덱스의 시작과 끝의 범위가 현재 분할된 리스트의 시작과 끝의 인덱스 범위를 벗어나는 경우 0을 반환한다(5번 줄).
* 위의 두 경우에 해당하지 않을 경우 분할된 두 리스트의 부분 합을 순환적으로 호출하여 계산한다(6~8번 줄).

1. Algorithm Query\_sum(seg\_tree, start, end, q\_s, q\_e, current)
2. Input: 세그먼트 트리 배열(seg\_tree), 분할된 리스트의 시작과 끝 인덱스(start,

end), 질의 인덱스(q\_s, q\_e), 현재 노드 인덱스(current)

1. Output: 질의 범위의 부분 합
2. if (q\_s <= start and q\_e >= end) return seg\_tree[current]
3. if (end < q\_s or start > q\_e) return 0
4. mid <- start + (end – start) / 2
5. child <- 2 \* current
6. return Query\_sum(seg\_tree, start, mid, q\_s, q\_e, child+1) +

Query\_sum(seg\_tree, mid+1, end, q\_s, q\_e, child+2)

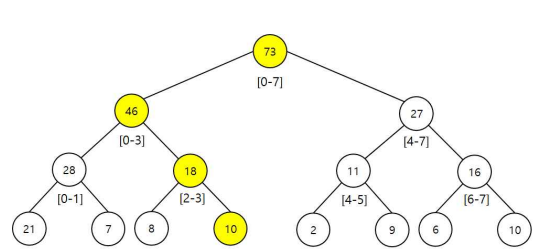
**(3) 세그먼트 트리의 갱신**

입력으로 주어진 리스트 배열 list[0], list[1], … , list[n-1]에서 list[i]에 해당하는 값이 value로 변경될 경우를 생각해 본다. 입력 리스트에서 이러한 변경이 일어나면 입력 리스트 배열과 세그먼트 트리 배열이 아래와 같이 갱신되어야 한다.

* list[i]를 변경된 값으로 대체한다. 즉, list[i] = value
* 세그먼트 트리 배열에서 list[i]의 단말 노드에 해당하는 모든 조상 노드들의 부분 합이 다시 계산되어야 한다.

예를 들어 (그림 1.2)의 세그먼트 트리는 입력으로 주어진 리스트 배열이 list[] = [21, 7, 8, 19, 2, 9, 6, 10]이다. 여기서 list[3]의 값을 10으로 변경하면 list 배열은 [21, 7, 8, 10, 2, 9, 6, 10]으로 변경되고 이와 동시에 세그먼트 트리도 갱신되어야 한다. (그림 1.3)은 (그림 1.2)에서 list[3]의 값이 10으로 변경되었을 때 갱신된 세그먼트 트리의 예를 보여준다.

(그림 1.3) list[3]의 값이 변경되었을 때 세그먼트 트리의 갱신



여기서 세그먼트 트리를 갱신할 때는, 변경된 값에서 원래의 값을 뺀 d\_value를 조상 노드들에 더해가면서 변경하게 된다.

앞의 내용을 적용한 (알고리즘 Segtree\_update)는 입력 리스트에서 인덱스 i에 해당하는 list[i] 값이 변경될 경우 세그먼트 트리를 갱신하는 알고리즘이다. 알고리즘의 입력으로 세그먼트 트리를 저장한 배열(seg\_tree), 분할 리스트의 시작과 끝의 위치(start/end), 변경된 위치의 인덱스(i), 변경된 위치에서 변경된 값과 원래 값의 차이 값(d\_value), 현재 위치의 인덱스(current)가 주어진다.

1. Algorithm Segtree\_update(seg\_tree, start, end, i, d\_value, current)
2. Input: 세그먼트 트리 배열(seg\_tree), 분할된 리스트의 시작과 끝의 인덱스(start,

end), 변경된 위치(i), 현재 노드의 인덱스(current)

1. Output: 갱신된 세그먼트 트리의 배열
2. if (i < start or i > end) return Null
3. seg\_tree[current] <- seg\_tree[current] + d\_value
4. if (start != end) {
5. mid <- start + (end – start) / 2
6. child <- 2 \* current
7. Segtree\_update((seg\_tree, start, mid, i, d\_value, child+1)
8. Segtree\_update((seg\_tree, mid+1, end, i, d\_value, child+2) }
9. end

4. 구현 및 실험 결과

프로그래밍 언어는 C++를 사용하여 구현하였고, cmath 라이브러리 함수를 사용했다. cmath 라이브러리는 트리를 구성할 때 트리의 높이를 구할 때 log 함수와 ceil 함수를 사용하여야 하고, 동적 할당을 위해 배열의 크기를 계산하여야 하므로 pow 함수를 사용해야 하므로 include 하였다. 구현은 알고리즘 분석 단계에서 썼던 슈도 코드를 C++ 문법 스타일로만 적용하여 바꾸었다. 실험을 위한 main.cpp 파일과 알고리즘의 핵심 함수들이 들어간 Segtree.cpp 파일과 Segtree.h 헤더 파일로 나누어져 있다. 전체 코드는 아래에 별도로 첨부하고, 여기서는 main.cpp 파일의 내용과 그 결과에 대해서만 나타내겠다.

(소스코드 1.4) main.cpp 파일 내용

|  |  |
| --- | --- |
| 1 | #include "Segtree.h" |
| 2 | #include <iostream> |
| 3 | using namespace std; |
| 4 |  |
| 5 | #define LIST\_LEN 6 |
| 6 |  |
| 7 | int main() { |
| 8 | int list[LIST\_LEN] = { 2,4,10,7,9,20 }; |
| 9 | int\* seg\_tree = Seg\_tree\_const(list, LIST\_LEN); |
| 10 | int q\_s, q\_e; |
| 11 | cout << "------리스트의 부분 합 질의-----\n"; |
| 12 | while (true) { |
| 13 | try { |
| 14 | cout << "시작 인덱스 : "; cin >> q\_s; |
| 15 | cout << "마지막 인덱스 : "; cin >> q\_e; |
| 16 | cout << "질의 범위의 부분 합 : " << Get\_query(seg\_tree, LIST\_LEN, q\_s, q\_e) << '\n'; |
| 17 | break; |
| 18 | } |
| 19 | catch (int) { |
| 20 | cout << "리스트의 범위를 벗어난 질의입니다.\n"; |
| 21 | } |
| 22 | } |
| 23 | int i, value; |
| 24 | cout << "----------리스트의 갱신---------\n"; |
| 25 | while (true) { |
| 26 | try { |
| 27 | cout << "수정할 인덱스 : "; cin >> i; |
| 28 | if (i < 0 || i > LIST\_LEN) throw i; |
| 29 | break; |
| 30 | } |
| 31 | catch (int) { |
| 32 | cout << "수정할 인덱스가 리스트의 범위를 벗어났습니다.\n"; |
| 33 | } |
| 34 | } |
| 35 | cout << "수정할 값 : "; cin >> value; |
| 36 | Segtree\_update(seg\_tree, 0, LIST\_LEN - 1, i, value - list[i], 0); |
| 37 | cout << "수정된 리스트의 질의 범위 부분 합 : " << Get\_query(seg\_tree, LIST\_LEN, q\_s, q\_e); |
| 38 | return 0; |
| 39 | } |

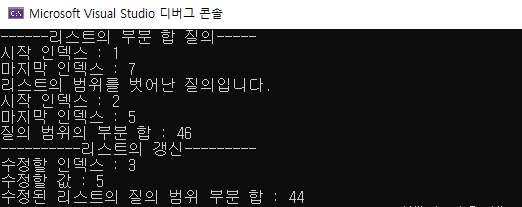
5번 줄에서 입력 리스트의 길이를 define으로 선언하여 길이가 다른 입력 리스트를 테스트할 때 쉽게 하였다. 8번 줄에서 입력 리스트를 넣고, 9번 줄에서 세그먼트 트리를 구성하는 함수를 호출하여 세그먼트 트리를 저장한다. 질의를 할 인덱스를 표준 입력을 받아 함수를 호출한다. Get\_query 함수에서 잘못된 질의에 대하여 오류 처리를 int형을 throw하는 것으로 처리하였으므로, while문과 try-catch문을 사용하여 Get\_query 함수에서 throw가 발생하면 리스트의 범위를 벗어났다는 문구를 출력한 후 다시 입력을 받도록 하였다. throw가 발생하지 않으면 자연스레 break가 실행되어 while문을 빠져나온다.

또한 수정할 인덱스도 비슷한 방식을 사용했다. 표준 입력을 받고, 입력받은 인덱스가 리스트의 범위를 벗어날 경우 Get\_query 함수처럼 int형을 throw하고, while문과 try-catch문을 사용하여 반복하여 입력받을 수 있도록 하였다.

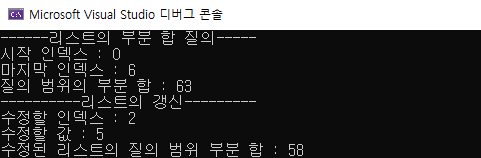
(사진 1.5)는 (소스코드 1.4)를 실행시킨 것으로 리스트의 범위를 벗어난 질의에 대한 오류 처리도 잘 되어있으며, 3번 인덱스의 원래 값인 7에서 5로 바뀌면서 부분합 또한 46에서 44로 2가 줄어든 것을 확인할 수 있다.

5번 줄과 8번 줄을 바꾸어 입력 리스트의 길이와 내용을 변경한 것들(사진 1.6, 사진 1.7)도 잘 실행되는 것을 아래 실험 결과 사진들로 확인할 수 있다.

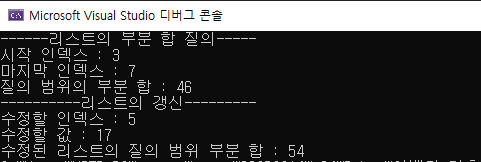
(사진 1.5) main.cpp 실행 결과



(사진 1.6) LIST\_LEN을 7로, 입력 list를 [2, 4, 10, 7, 9, 20, 11]로 했을 때의 실행 결과



(사진 1.7) LIST\_LEN을 8로, 입력 list를 [21, 7, 8, 19, 2, 9, 6, 10]로 했을 때의 실행 결과



5. 기타 사항

시간 복잡도를 구할 때는 O(log2n)과 O(logn)의 차이가 없으므로 편의상 logn으로 이 보고서의 내용을 작성하였으나 구현 시에는 이를 주의하지 않으면 값이 달라질 수 있으므로 주의하도록 한다.

[붙임] 소스코드

▪ main.cpp

|  |  |
| --- | --- |
| 1 | #include "Segtree.h" |
| 2 | #include <iostream> |
| 3 | using namespace std; |
| 4 |  |
| 5 | #define LIST\_LEN 6 |
| 6 |  |
| 7 | int main() { |
| 8 | int list[LIST\_LEN] = { 2,4,10,7,9,20 }; |
| 9 | int\* seg\_tree = Seg\_tree\_const(list, LIST\_LEN); |
| 10 | int q\_s, q\_e; |
| 11 | cout << "------리스트의 부분 합 질의-----\n"; |
| 12 | while (true) { |
| 13 | try { |
| 14 | cout << "시작 인덱스 : "; cin >> q\_s; |
| 15 | cout << "마지막 인덱스 : "; cin >> q\_e; |
| 16 | cout << "질의 범위의 부분 합 : " << Get\_query(seg\_tree, LIST\_LEN, q\_s, q\_e) << '\n'; |
| 17 | break; |
| 18 | } |
| 19 | catch (int) { |
| 20 | cout << "리스트의 범위를 벗어난 질의입니다.\n"; |
| 21 | } |
| 22 | } |
| 23 | int i, value; |
| 24 | cout << "----------리스트의 갱신---------\n"; |
| 25 | while (true) { |
| 26 | try { |
| 27 | cout << "수정할 인덱스 : "; cin >> i; |
| 28 | if (i < 0 || i > LIST\_LEN) throw i; |
| 29 | break; |
| 30 | } |
| 31 | catch (int) { |
| 32 | cout << "수정할 인덱스가 리스트의 범위를 벗어났습니다.\n"; |
| 33 | } |
| 34 | } |
| 35 | cout << "수정할 값 : "; cin >> value; |
| 36 | Segtree\_update(seg\_tree, 0, LIST\_LEN - 1, i, value - list[i], 0); |
| 37 | cout << "수정된 리스트의 질의 범위 부분 합 : " << Get\_query(seg\_tree, LIST\_LEN, q\_s, q\_e); |
| 38 | return 0; |
| 39 | } |

▪ Segtree.h

|  |  |
| --- | --- |
| 1 | #pragma once |
| 2 | int\* Seg\_tree\_const(int\* list, int n); |
| 3 | int Construct\_tree(int\* list, int start, int end, int\* seg\_tree, int current); |
| 4 | int Get\_query(int\* seg\_tree, int n, int q\_s, int q\_e); |
| 5 | int Query\_sum(int\* seg\_tree, int start, int end, int q\_s, int q\_e, int current); |
| 6 | int\* Segtree\_update(int\* seg\_tree, int start, int end, int i, int d\_value, int current); |

▪ Segtree.cpp

|  |  |
| --- | --- |
| 1 | #include "Segtree.h" |
| 2 | #include <cmath> |
| 3 |  |
| 4 | int\* Seg\_tree\_const(int\* list, int n) { |
| 5 | int height = (int)ceil(log2(n)); //높이 |
| 6 | int t\_size = 2 \* (int)pow(2, height) - 1; //배열의 크기 |
| 7 | int\* seg\_tree = new int[t\_size]; |
| 8 | Construct\_tree(list, 0, n - 1, seg\_tree, 0); |
| 9 | return seg\_tree; |
| 10 | } |
| 11 | int Construct\_tree(int\* list, int start, int end, int\* seg\_tree, int current) { |
| 12 | if (start == end) { |
| 13 | seg\_tree[current] = list[start]; |
| 14 | return list[start]; |
| 15 | } |
| 16 | int mid = (start + end) / 2; |
| 17 | int child = 2 \* current; |
| 18 | seg\_tree[current] = Construct\_tree(list, start, mid, seg\_tree, child + 1)  + Construct\_tree(list, mid + 1, end, seg\_tree, child + 2); |
| 19 | return seg\_tree[current]; |
| 20 | } |
| 21 | int Get\_query(int\* seg\_tree, int n, int q\_s, int q\_e) { |
| 22 | if (q\_s < 0 || q\_e > n - 1 || q\_s > q\_e) { |
| 23 | throw 0; |
| 24 | } |
| 25 | return Query\_sum(seg\_tree, 0, n - 1, q\_s, q\_e, 0); |
| 26 | } |
| 27 | int Query\_sum(int\* seg\_tree, int start, int end, int q\_s, int q\_e, int current) { |
| 28 | if (q\_s <= start && q\_e >= end) { //질의 범위에 완전히 포함되는 노드 |
| 29 | return seg\_tree[current]; |
| 30 | } |
| 31 | if (end < q\_s || start > q\_e) { //질의 범위를 벗어날 때 |
| 32 | return 0; |
| 33 | } |
| 34 | int mid = (start + end) / 2; |
| 35 | int child = 2 \* current; |
| 36 | return Query\_sum(seg\_tree, start, mid, q\_s, q\_e, child + 1)  + Query\_sum(seg\_tree, mid + 1, end, q\_s, q\_e, child + 2); |
| 37 | } |
| 38 | int\* Segtree\_update(int\* seg\_tree, int start, int end, int i, int d\_value, int current) { |
| 39 | if (i < start || i > end) { |
| 40 | return NULL; |
| 41 | } |
| 42 | seg\_tree[current] += d\_value; |
| 43 | if (start != end) { |
| 44 | int mid = (start + end) / 2; |
| 45 | int child = 2 \* current; |
| 46 | Segtree\_update(seg\_tree, start, mid, i, d\_value, child + 1); |
| 47 | Segtree\_update(seg\_tree, mid + 1, end, i, d\_value, child + 2); |
| 48 | } |
| 49 | return seg\_tree; |
| 50 | } |